Expenses

Welcome to Expenses on Exercism's Go Track. If you need help running the tests or submitting your code, check out `HELP.md`. If you get stuck on the exercise, check out `HINTS.md`, but try and solve it without using those first :)

Introduction

In Go, functions are first-class values. This means that you can do with functions the same things you can do with all other values - assign functions to variables, pass them as arguments to other functions or even return functions from other functions.

Below we are creating two functions, `engGreeting` and `espGreeting` and we are assigning them to the variable `greeting`:

```
import "fmt"

func engGreeting(name string) string {
        return fmt.Sprintf("Hello %s, nice to meet you!", name)
}

func espGreeting(name string) string {
        return fmt.Sprintf("¡Hola %s, mucho gusto!", name)
}

greeting := engGreeting                 // greeting is a variable of type func(string) st
fmt.Println(greeting("Alice"))  // Hello Alice, nice to meet you!

greeting = espGreeting
fmt.Println(greeting("Alice"))  // ¡Hola Alice, mucho gusto!
```

Function values provide an opportunity to parametrize functions not only with data but with behavior too. In the following example, we are passing behaviour to the `dialog` function via the `greetingFunc` parameter:

```
func dialog(name string, greetingFunc func(string) string) {
        fmt.Println(greetingFunc(name))
        fmt.Println("I'm a dialog bot.")
```

```
}

func espGreeting(name string) string {
        return fmt.Sprintf("¡Hola %s, mucho gusto!", name)
}

greeting := espGreeting
dialog("Alice", greeting)
// =>
// ¡Hola Alice, mucho gusto!
// I'm a dialog bot.
```

The value of an uninitialized variable of function type is `nil`. Therefore, calling a `nil` function value causes a panic.

```
var dutchGreeting func(string) string
dutchGreeting("Alice") // panic: call of nil function
```

Function values can be compared with `nil`. This can be useful to avoid unnecessary program panics.

```
var dutchGreeting func(string) string
if dutchGreeting != nil {
        dutchGreeting("Alice") // safe to call dutchGreeting
}
```

Function types

Using function values is possible thanks to the function types in Go. A function type denotes the set of all functions with the same sequence of parameter types and the same sequence of result types. User-defined types can be declared on top of function types. For instance, the `dialog` function from the previous examples can be updated as following:

```
type greetingFunc func(string) string

func dialog(name string, f greetingFunc) {
        fmt.Println(f(name))
        fmt.Println("I'm a dialog bot.")
}
```

Anonymous functions

Another powerful tool that is available thanks to first-class functions support is anonymous functions. Anonymous functions are defined at their point of use, without a name following the `func` keyword. Such functions have access to the variables of the enclosing function.

For example:

```
func fib() func() int {
        var n1, n2 int

        return func() int {
                if n1 == 0 && n2 == 0 {
                        n1 = 1
                } else {
                        n1, n2 = n2, n1 + n2
                }
                return n2
        }
}

next := fib()
for i := 0; i < N; i++ {
  fmt.Printf("F%d\t= %4d\n", i, next())
}
```

A call to `fib` declares the variables `n1` and `n2` and returns an anonymous function that, in turn, changes the values of these variables each time the function is called. Nth calls of the anonymous function return the Nth number of the Fibonacci sequence starting from 0. The anonymous inner function has access to the local variables (`n1` and `n2`) of the enclosing function `fib` . This is a great way to have function values keep state between calls. We say that the anonymous function is a closure of the variables `n1` and `n2` . Closures are widely used in programming and you might see other languages supporting them.

Instructions

Bob is a financial adviser and helps people to manage their expenses. Bob's clients send expenses records for him to analyze. Bob has records for the previous periods so he can see changes in spending. Bob does not like calendars and uses *Bob epoch* instead of dates. Bob epoch is the number of days elapsed since Bob's client started their activity.

In this exercise, you are going to build a program to help Bob manage and analyze the expenses of his clients.

Bob works with `Record`s and `DaysPeriod`s.

A `Record` represents an expense record that contains the day in which the expense was made, the money spent, and the category of the expense.

```
// Record represents an expense record.
type Record struct {
        Day         int
        Amount      float64
        Category    string
}
```

A `DaysPeriod` represents a range of days and includes all days from the day `From` up to the day `To`. Both ends are included in the range.

```
// DaysPeriod represents a period of days.
type DaysPeriod struct {
        From int
        To   int
}

p := DaysPeriod{From: 1, To: 31}
// p represents all days from the day 1 to the day 31:
//  - days 1, 20, 16 and 31 are examples of days that are included
//    in the range of time specified by p
//  - days 50 and 40 are examples of days that are not included
//    in the range of time specified by p
```

1. Implement a general records filter

Bob deals with a lot of records every day, but not all of them are interesting depending on the analysis Bob is making. Let's help Bob perform some basic filtering of records.

Implement the generic `Filter` function to filter records according to a criteria given by a function. This filter function accepts a collection of records and a predicate function and returns only the records in the collection that satisfy the predicate.

```
records := []Record{
  {Day: 1, Amount: 15, Category: "groceries"},
  {Day: 11, Amount: 300, Category: "utility-bills"},
  {Day: 12, Amount: 28, Category: "groceries"},
}

// Day1Records only returns true for records that are from day 1
func Day1Records(r Record) bool {
  return r.Day == 1
}

Filter(records, Day1Records)
// =>
// [
//   {Day: 1, Amount: 15, Category: "groceries"}
// ]
```

2. Filter records within a period of time

Bob frequently needs to filter records that are in a given period of time.

Implement the `ByDaysPeriod` function that will help Bob create such filters. This function accepts a `DaysPeriod` and returns function that takes a record and tells whether the record is in the period of time specified by the `DaysPeriod` given as argument.

```
records := []Record{
  {Day: 1, Amount: 15, Category: "groceries"},
  {Day: 11, Amount: 300, Category: "utility-bills"},
  {Day: 12, Amount: 28, Category: "groceries"},
  {Day: 26, Amount: 300, Category: "university"},
  {Day: 28, Amount: 1300, Category: "rent"},
}

period := DaysPeriod{From: 1, To: 15}

Filter(records, ByDaysPeriod(period))
// =>
// [
//   {Day: 1, Amount: 15, Category: "groceries"},
//   {Day: 11, Amount: 300, Category: "utility-bills"},
//   {Day: 12, Amount: 28, Category: "groceries"},
// ]
```

3. Filter records by category

Other than filtering records by a period of time, Bob also needs to filter records by its category.

Implement the `ByCategory` function that will help Bob create such filters. This function accepts a category and returns a function that takes a record and tells whether the category of this record is the same as the category given as the argument.

```
records := []Record{
  {Day: 1, Amount: 15, Category: "groceries"},
  {Day: 11, Amount: 300, Category: "utility-bills"},
  {Day: 12, Amount: 28, Category: "groceries"},
  {Day: 28, Amount: 1300, Category: "rent"},
}

Filter(records, ByCategory("groceries"))
// =>
// [
//   {Day: 1, Amount: 15, Category: "groceries"},
//   {Day: 12, Amount: 28, Category: "groceries"},
// ]
```

4. Calculate the total amount of expenses in a period

Bob also wants to know the total amount of the expenses in a period of time.

Implement the `TotalByPeriod` function to return a sum of expenses in the days period.

```
records := []Record{
  {Day: 15, Amount: 16, Category: "entertainment"},
  {Day: 32, Amount: 20, Category: "groceries"},
  {Day: 40, Amount: 30, Category: "entertainment"}
}

p1 := DaysPeriod{From: 1, To: 30}
p2 := DaysPeriod{From: 31, To: 60}

TotalByPeriod(records, p1)
// => 16

TotalByPeriod(records, p2)
```

```
// => 50
```

5. Calculate the total expenses for records of a category in a period

For the most complex reports Bob makes to his clients, Bob needs to filter records by category and period of time at the same time. That means Bob wants to know the total expenses for records in a category in a given period of time.

Implement the `CategoryExpenses` function that returns the total amount of expenses in a category in a given period of days. The function should also differentiate the case when the given category is not present in the expenses records and the case when there are no category's expenses in the provided period. When the category is not a category of any of the records (regardless of period of time) the function should return an error.

```
p1 := DaysPeriod{From: 1, To: 30}
p2 := DaysPeriod{From: 31, To: 60}

records := []Record{
  {Day: 1, Amount: 15, Category: "groceries"},
  {Day: 11, Amount: 300, Category: "utility-bills"},
  {Day: 12, Amount: 28, Category: "groceries"},
  {Day: 26, Amount: 300, Category: "university"},
  {Day: 28, Amount: 1300, Category: "rent"},
}

CategoryExpenses(records, p1, "entertainment")
// => 0, error(unknown category entertainment)

CategoryExpenses(records, p1, "rent")
// => 1300, nil

CategoryExpenses(records, p2, "rent")
// => 0, nil
```

Source

Created by

- @antklim
- @andrerfcsantos